International Journal of Information, Control and Computer Sciences
ISSN: 2517-9942
Vol:9, No:11, 2015

Cognitive Weighted Polymorphism Factor: A Comprehension Augmented Complexity Metric

T. Francis Thamburaj, A. Aloysius

***Abstract*—**Polymorphism is one of the main pillars of object-oriented paradigm. It induces hidden forms of class dependencies which may impact software quality, resulting in higher cost factor for comprehending, debugging, testing, and maintaining the software. In this paper, a new cognitive complexity metric called Cognitive Weighted Polymorphism Factor (CWPF) is proposed. Apart from the software structural complexity, it includes the cognitive complexity on the basis of type. The cognitive weights are calibrated based on 27 empirical studies with 120 persons. A case study and experimentation of the new software metric shows positive results. Further, a comparative study is made and the correlation test has proved that CWPF complexity metric is a better, more comprehensive, and more realistic indicator of the software complexity than Abreu's Polymorphism Factor (PF) complexity metric.

***Keywords*—**Cognitive complexity metric, cognitive weighted polymorphism factor, object-oriented metrics, polymorphism factor, software metrics.

I. Introduction

POLYMORPHISM is one of the main features of object-oriented paradigm. The other salient features of object oriented paradigm are information hiding, inheritance, cohesion, and coupling between objects. The term 'polymorphism' in Greek means many forms. In the object-oriented programming, polymorphism refers to the ability of a programming language to process objects differently depending on their data type or class. More specifically, it is the ability to redefine or implement methods or interfaces in derived classes. Polymorphism has been assured to improve extensibility and reusability technique [1].

The importance of polymorphism stems from the various benefits it offers in the software development life-cycle, to bring out qualitatively efficient and effective software. The main advantage of polymorphism is that it gives a high degree of freedom and decoupling because the implementation behind an interface is hidden from the clients. At coding time, you only have to worry about programming to the interface. At test time, you can naively substitute a mock object and validate your unit of code independently. At run time, you can dynamically supply different implementations based on application logic in order to achieve late binding. Another important benefit of polymorphism is scaling the complexity of the software. Polymorphism is a critical and central mechanism for scaling a system's complexity, because it allows abstraction and economies of scale in client/server contracts, and also allows for the sane extension of functionality, because it always allows to add new implementations over time. Yet another advantage is due to parametric polymorphism. It allows a function or a data type to be written generically, so that it can handle different types of values uniformly without depending on their type. Parametric polymorphism is a way to make a language more expressive, while still maintaining full static type-safety.

Polymorphism, based on the method binding nature, can be classified into 3 types. 1) *Pure Polymorphism*: It is achieved by the evocation of the same member function name with different signatures inside the one and the same class scope. The signature consists of the number, the type and the order of the arguments. It is equal to method overloading within a class. It is also called as parametric overloading. 2) *Static Polymorphism*: It is composed level polymorphism that occurs in class hierarchical tree. It is equivalent to method overriding. It is achieved by defining the member functions in different classes with the same name but with different signatures. These classes may or may not be linked by inheritance relations. For both pure and static polymorphisms, the method binding happens at the compile time. 3) *Dynamic Polymorphism*: It is also composed level polymorphism and equivalent to method overriding. It is achieved by defining the member functions in child classes with the same name and same signature unlike static polymorphism. It is the ability to use the same name and the same signature in an overridden method [2]. This research article focuses only on the different forms of polymorphism available in Java language.

Due to the plethora of benefits, polymorphism concept is considered as one of the key concerns in determining the quality of object-oriented design [3]. However, there are some polymorphic forms, when used in combination with inheritance, can penetrate encapsulation boundaries and create hidden dependencies which affects the quality of the software [2]. Hence, there are many benefits as well as problems in the use of polymorphism. This calls for cautious and correct measure of polymorphism usage. In order to measure the usage of polymorphism and the software quality with respect to usage of polymorphism, the polymorphism metric is used. It measures the degree of method overriding within the class or in the class inheritance tree. Abreu et al. recommends the interval design heuristic for the polymorphic complexity. According to him, the polymorphism factor complexity metric value has to be typically kept in the middle range and preferably around 10%, which decreases the defect density as

T. Francis Thamburaj, Assistant Professor and A. Aloysius, Assistant Professor are with the Department of Computer Science, St. Joseph's College (Autonomous), Bharathidasan University, Thiruchchiraappalli – 620 002, Tamil Nadu, India (Phone: +91 9442609111, +91 9443399227; e-mail: francisthamburaj@gmail.com, aloysius1972@gmail.com).

well as rework. [4]. Thus, the use of polymorphism has to be in the golden threshold. When the polymorphism factor is high in a software system, the comprehensibility, modifiability, testability, and maintainability become harder and costlier. On the other hand, when it is very low, the reusability and extensibility become negligible. Hence, there is a greater need to measure the complexity of polymorphism more accurately.

II. Literature Survey

Several software object-oriented metrics have been proposed in the past [5]–[10]. Only a few of the proposed object-oriented design metrics have focused on polymorphism. In particular, Abreu et al. have proposed Polymorphism Factor (PF) metric in their metric suite called Metrics for Object Oriented Design (MOOD). PF is the number of methods that redefine (overrides) inherited methods, divided by maximum number of possible distinct polymorphic situations [7]. Lorenz and Kidd proposed a polymorphism measure called NMO in their metric suite. It refers to the number of methods overridden by a single subclass [8]. NMO is a class-level metric while PF is a system-level metric, which measures the degree of method overriding in the whole type tree [11]. Benlarbi et al. proposed a set of 5 polymorphic measures based on the static/dynamic polymorphism forms with simple inheritance relationships, not including the multiple inheritance or friendship relations. They are overloading in stand-alone classes (OVO), Static Polymorphism in Ancestors (SPA), Static Polymorphism in Descendants (SPD), Dynamic Polymorphism in Ancestors (DPA), Dynamic Polymorphism in Descendants (DPD) [2]. Bansiya, in his QMOOD metric suite, proposed the polymorphic metric NOP. It is the count of the methods that can exhibit polymorphic behavior [10]. The complexity metric NOP refers to the virtual methods in C++. In Java, as this research article focuses only on Java language, all non-static methods are by default virtual functions. Only methods marked with the keyword final, which cannot be overridden, along with private methods, which are not inherited, are non-virtual. In the area of dynamic metrics for polymorphism, Dufour et al. developed as many as 17 dynamic metrics for polymorphism using a framework for Java [12]. Sandhu et al. give a set of 11 dynamic metrics for polymorphism in object oriented system [13].

All the polymorphism metrics proposed have only considered the architectural complexity. None of them has dealt with cognitive complexity of polymorphism. Wang observed that the traditional measurements cannot actually reflect the real complexity of software systems in a software design, representation, cognition, comprehension and maintenance. Instead, the cognitive complexity metrics is an ideal measure of software functional complexities and sizes, as it represents the real semantic complexity by integrating both the operational and architectural complexities [14]. The cognitive complexity is defined as the mental burden on the user who deals with the code as developer, tester, maintainer etc. It is measured in terms of cognitive weights. Cognitive weights are defined as the extent of difficulty or relative time and effort required for comprehending given software, and measure the complexity of logical structure of software [15]. Only a few cognitive complexity metrics are proposed in object-oriented paradigm. Aloysius et al. proposed cognitive complexity metrics for Coupling between Object, Response For a Class, etc., but not for polymorphism [15], [16]. Hence, there is a need to propose cognitive complexity based polymorphism metric.

The proposed metric CWPF is explained in Section III, the calibration of cognitive weights is discussed in Section IV, the experimentation and case study of the new metric is described in Section V, the comparative study of CWPF with PF is done in Section VI, the normalized CWPF is portrayed in Section VII, and Section VIII presents the conclusion and the possible future works.

III. Proposed Metric: Cognitive Weighted Polymorphism Factor

Abreu et al. measure the software complexity due to the presence of polymorphism by dividing the actual polymorphism present in the system with the hypothetically possible maximum polymorphism potential if all the methods are overridden in all classes except the base ones. Here, he considers the polymorphism from the perspective of hierarchical inheritance tree. He defines the PF complexity metric as a quotient ranging from 0% to 100% in order to compare and derive conclusions among heterogeneous systems with different sizes, complexities, application domains and implementation languages [17]. In the PF quotient, the numerator represents actual number of methods that redefine the inherited method either at the compile time (statically) or at the run time (dynamically) and the denominator represents the total maximum number of possible different polymorphic situations in the whole software system. Formally, the PF complexity metric is given by (1):

$$PF = \frac{\sum_{i=1}^{TC} M_o(C_i)}{\sum_{i=1}^{TC}[M_n(C_i)*DC(C_i)]} \quad (1)$$

where, $M_o(C_i)$ = number of overriding methods in class C_i, $M_n(C_i)$ = number of new methods defined in class C_i, $DC(C_i)$ = number of children for class C_i, TC = Total number of Classes in the whole software system.

The proposed Cognitive Weighted Polymorphism Factor attaches different cognitive weights based on the above mentioned three categories of polymorphism according to [2]. The proposed CWPF calculates not only the architectural complexity of the polymorphism, but also the cognitive complexity arising from the effort needed to comprehend different types of polymorphism involved in the software system under consideration. The formal mathematical definition of CWPF is given in (2):

$$CWPF = \frac{\sum_{i=1}^{TC} CWM_o(C_i)}{\sum_{i=1}^{TC}[M_n(C_i)*DC(C_i)]*ACW} \quad (2)$$

where, $CWM_o(C_i) = N_{PP} * CW_{PP} + N_{SP} * CW_{SP} + N_{DP} * CW_{DP}$, $M_n(C_i)$ = number of overriding methods in class C_i, $DC(C_i)$ = number of children for class C_i, TC = Total number of Classes.

$$ACW = (CW_{PP} + CW_{SP} + CW_{DP}) / 3$$

where, N_{PP} = number of pure polymorphism, N_{SP} = number of static polymorphism, N_{DP} = number of dynamic polymorphism, CW_{PP} = cognitive weight of pure polymorphism, CW_{SP} = cognitive weight of static polymorphism, CW_{DP} = cognitive weight of dynamic polymorphism.

In the numerator of (2), each type of polymorphism is multiplied with the corresponding cognitive weight. The denominator is multiplied by ACW the mean of the three cognitive weights.

IV. Calibration of Cognitive Weights

In this section, cognitive weights for Pure Polymorphism (PP), Static Polymorphism (SP), and Dynamic Polymorphism (DP) are calibrated separately. In order to find the cognitive weight factor for each of the three types of polymorphism, a comprehension test was conducted for three different groups of students to find out the time taken to understand the complexity of different types of polymorphism. These groups of students had sufficient exposure to Java programming and especially, in understanding various types of polymorphism. Around 40 students, who have scored 65% and above marks in Semester examination, were selected in each group. One undergraduate group and two postgraduate groups are called for the comprehension test and supplied 9 different programs namely, P1 to P9, three for each type of polymorphism with multiple choice answers. The time taken by each student to understand the program and to choose the best answer was recorded after the completion of each program. This process is repeated for each group of students.

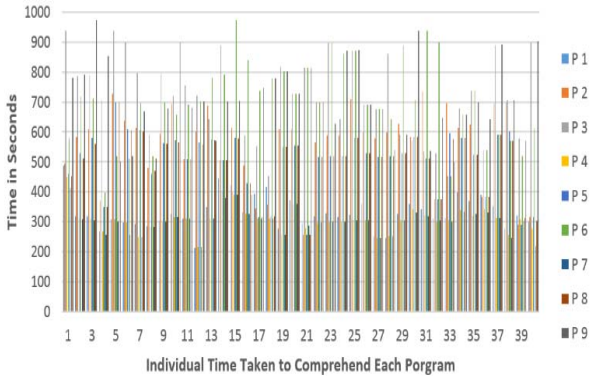

Fig. 1 Sample Individual Comprehension Time Chart

To be accurate, these program comprehension tests were conducted online and the comprehension timings were registered automatically by the computer in seconds. A sample of the individual time taken to comprehend each of the nine programs from P1 to P9 is graphically shown in Fig. 1.

The average time taken to comprehend each individual program from P1 to P9 by each group was calculated, so as to get 27 different Comprehension Mean Times (CMT). Since 3 different groups of students have done the comprehension test for the same program, their values are averaged to obtain the 9 different values in the CMT column of Table I. The tested programs and the corresponding CMT values are grouped into PP, SP, and DP categories. Then, the average of each category is calculated and displayed in the last column of Table I as the average CMT in seconds.

In Fig. 2, which is the pictorial representation of Table I, the CMTs for each type of polymorphism are grouped to gather under the headings of Pure, Static, and Dynamic in order to bring out the group differences in comprehending the programs. The undergraduate group, represented by Group 1, has taken more time to comprehend than the postgraduate students, represented by Group 2 and Group 3. This fact is intuitively valid indeed, as the UG students had less exposure to Java than the PG students and hence more meaningful [18]. The average CMT for each category of polymorphism is displayed as value above each group of bars.

TABLE I
Calibration of Cognitive Weights

Category	Program #	CMT (Secs)	Average CMT (Secs)
Pure Polymorphism (PP)	P1	330.40	313.06
	P2	307.05	
	P3	301.73	
Static Polymorphism (SP)	P4	556.13	520.43
	P5	503.93	
	P6	501.23	
Dynamic Polymorphism (DP)	P7	720.20	707.81
	P8	702.18	
	P9	701.05	

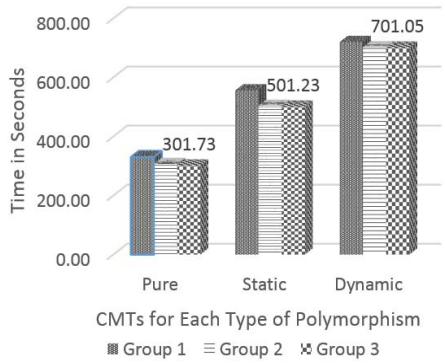

Fig. 2 Categorized Cognitive Weights

From Figs. 1 and 2, Table I, it is clear that the average CMT to understand the static polymorphism is more than the pure polymorphism and the time taken for the dynamic polymorphism is the highest. This implies that the cognitive load to understand the DP is greater than SP and SP is greater than PP. Finally, the cognitive weights are got by the MOD 100 function and the cognitive value for PP is 3, and for the SP is 5, and for the DP is 7. The ratio of these values correspond to our natural intuitive understanding of difficulties and hence more meaningful and truthful [17].

V. Experimentation and Case Study

The proposed CWPF metric given by (2) is evaluated with the following case study program. The program has five classes. The root class has four methods. The C2 class has two statically overridden methods. The C3 class has two dynamically overridden methods and two new methods. The C4 and C5 classes have no methods. The UML diagram of the program is given in Fig. 3.

Applying Abreu's metric as given in (1),

$$PF = \frac{(C1)+M_o(C2)+M_o(C3)+M_o(C4)+M_o(C5)}{M_n(C1)*4+M_n(C2)*0+M_n(C3)*2+M_n(C4)*0+M_n(C5)*0}$$

= (0+2+2+0+0) / (4*4 + 0 + 2*2 + 0 + 0)
= 4/20 or 0.20 or 20%

```
1:  /*** Case Study Program ***/
2:  class C1 {
3:      float v1 = 3;
4:      void m1(int i){ }
5:      void m2(char ch){ }
6:      float m3(){
7:          return 4 * v1;}
8:      float m4(){
9:          return v1 * v1;}
10: }
11: class C2 extends C1 {
12:     void m1(float f) { }
13:     void m2(String s) { }
14: }
15: class C3 extends C1 {
16:     float m3(){
17:         return 2 * 3.14 * v1;}
18:     float m4(){
19:         return v1 * v1 * v1;}
20:     int m5(int a, int b) {
21:         return a+b;}
22:     int m6(int a, int b) {
23:         return a * b;}
24: }
25: class C4 extends C3 {
26: }
27: class C5 extends C3 {
28: }
```

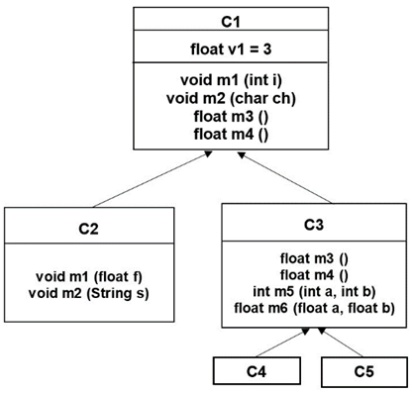

Fig. 3 UML Diagram of Case Study Program

Applying newly proposed metric as given in (2),

$$CWPF = \frac{M_o(C1)+M_o(C2)*CW_{sp}+M_o(C3)*CW_{dp}+M_o(C4)+M_o(C5)}{\left(\begin{matrix}M_n(C1)*4+M_n(C2)*0+M_n(C3)*2\\ +M_n(C4)*0+M_n(C5)*0\end{matrix}\right)*ACW}$$

= (0 + 2*5 + 2*7 + 0 + 0) / ((4*4 + 0 + 2*2 + 0 + 0)*5)
= 24/100 or 0.24 or 24%

In the CWPF calculation, each overriding is multiplied by the corresponding cognitive weights in the numerator and the denominator is multiplied by the average cognitive weight.

VI. Comparative Study

Comparative studies are done to validate complexity metric [19]. So, a comparative study has been made with PF complexity metric which is in the most widely accepted and empirically verified MOOD metric suite [7]. While Abreu, in proposing the Polymorphism Factor metric, did not provide the total complexity of the class by considering the cognitive complexity due to polymorphism in that class. This is precisely the difference between the PF and CWPF metrics. The CWPF metric is more advanced than the PF of Abreu, since it includes the cognitive complexity that arises due to various types of polymorphisms by calibrating and attaching the cognitive weights appropriately. The polymorphic cognitive weights are the effort needed by the programmer or the user to understand the different types of polymorphism embedded in the program.

In order to compare the proposed CWPF metric with the already existing PF metric, a comprehension test was conducted to a group of students who are doing their master's degree. There were forty students in the group who participated in the test. The students were given five different programs, P1 to P5, in Java for the comprehension test. The time taken to complete the test in seconds is captured in the online style, in order to maintain the accuracy. The average time taken to comprehend each program by all students is calculated and placed in Table II under the column head CMT. The PF and CWPF values are calculated manually for each of the five programs.

TABLE II
Complexity Metric Values and CMT Values

Program #	PF	CWPF	CMT (Secs)
P1	8.3	11.7	315.2143
P2	7.69	9.2	274.5714
P3	12	15.2	362.8571
P4	10.3	11.7	352.1429
P5	9.52	9.52	259.9286

The polymorphism factor complexity of the class is calculated by computing static polymorphism (SP), dynamic polymorphism (DP), and pure polymorphism (PP). This is better indicator than the simple PF of Abreu. The weight of each type of polymorphism factor is calculated by using cognitive weights. The weighting factor of each type is calculated similar to that is suggested by [14]. It is found that the resulting value of CWPF is larger than PF since, in PF, the weight of each type of polymorphism is assumed to be one. However, the calculation of the CWPF is more realistic

because it includes cognitive complexity of polymorphism. The obtained values of PF, CWPF and the CMT of the five different programs are tabulated in Table II.

A correlation analysis was performed between PF and CMT and also between CWPF and CMT values. Both correlations were found to be positive, which means that both values of PF and CWPF correlates well with CMT values found in the empirical test conducted. The Pearson correlation r (PF, CMT) is 0.7146 and r (CWPF, CMT) is 0.8836. The bigger correlation value for CWPF than the PF concludes that CWPF is a better indicator of complexity of the classes with various types of polymorphism. This fact is further clarified clearly in the correlation chart given in Fig. 4. The CWPF values are closer to the actual mean time taken by the students to understand the complexity of different types of polymorphism in the given programs than the values of PF. Thus the proposed CWPF complexity metric, as it includes the cognitive complexity, is proved to be more robust and more realistic complexity metric than PF complexity metric which considers only the architectural complexity.

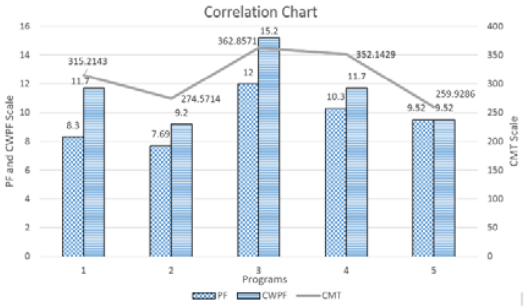

Fig. 4 Comparison of PF and CWPF with CMT

VII. Normalized CWPF

The calculation of the complexity values of CWPF with (2) is mainly meant for the comparative study with Abreu's PF. Here, the complexity values for CWPF will be in the range of 0 to 1.4. When there is no polymorphism, complexity value of CWPF becomes zero. When all the methods are overridden with pure polymorphism, the complexity value of CWPF will be 0.6. When all the methods are overridden with static polymorphism, the complexity value will be 1. When all the methods are overridden with dynamic polymorphism, the complexity value will be 1.4. Thus, the range of complexity values goes from 0 to 1.4.

To make the complexity values of CWPF to lie in the range of 0 to 1, normalization can be done, by replacing ACW by Maximum Cognitive Weight (MCW) in (2), as given here.

$$MCW = Max\ (CW_{PP}, CW_{SP}, CW_{DP})$$

Then, the complexity values, when all the methods are overridden with PP, SP, DP, will be 0.428, 0.714, and 1 respectively, whereas no polymorphism case will have 0 complexity value.

According to the normalized CWPF, the CWPF values of Table II will change as given in Table III.

The corresponding graphical representation of the normalized complexity values of CWPF and the CMT is shown in Fig. 5.

TABLE III
Complexity Metric Values and CMT Values

Program #	CWPF	CMT (Secs)
P1	8.3	315.2143
P2	6.6	274.5714
P3	10.9	362.8571
P4	8.37	352.1429
P5	6.8	259.9286

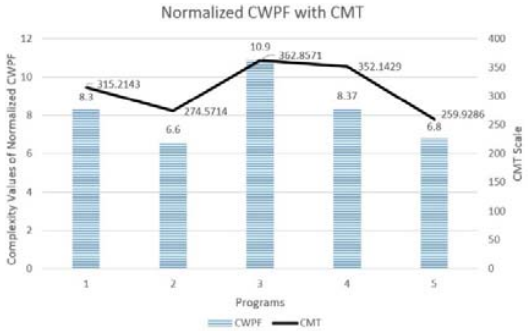

Fig. 5 Normalized CWPF with CMT

The Pearson correlation coefficient for the PF and CMT is 0.7146 and for CWPF and CMT is 0.8836. Hence, CWPF is statically proved to be the better measure than the PF measure. After the normalization of CWPF, the correlation coefficient between the CWPF and CMT is again 0.8833. Thus, the use of MCW adjusts the complexity values to lie in the range of 0 to 1, while maintaining the same correlation coefficient. This new range of cognitive complexity values aligns the scale of complexity values of CWPF with that of Abreu's complexity metric due to polymorphism [7]. According to him, the denominator represents the maximum number of possible distinct usage of the polymorphism and the purpose of the denominator is to act as normalizer for the complexity metric PF [20]. Therefore, it will be more apt and meaningful to multiply the denominator of the complexity metric CWPF with the maximum possible cognitive weight in order to act as normalizer as far as the cognitive complexity metric is concerned. Further, the normalized complexity metric CWPF becomes dimensionless satisfying one of the seven criteria of Abreu's for a good object-oriented metric [7].

VIII. Conclusion and Future Works

A new complexity metric called Cognitive Weighted Polymorphic Factor has been proposed and formulated for measuring the class level complexity. The polymorphism factor given by Abreu measures only the structural complexity. The cognitive weighted polymorphism factor captures not only the structural complexity, but also the cognitive complexity. The new polymorphism complexity metric is calibrated using series of comprehension tests and

found that the cognitive load for different types of polymorphism differ in the increasing order from pure, static, and dynamic. The new polymorphism complexity metric CWPF is more comprehensive in nature and more true to reality. This is proved by case study. Further, this is confirmed empirically by conducting a set of comprehension test and performing the correlation analysis that concluded saying that the CWPF is a better indicator of class complexity than the PF. The normalization of CWPF has made the complexity metric more robust as it becomes dimensionless, satisfying the Aberu's criteria for good object-oriented metric.

Regarding the future works, a tool has to be developed for calculating the CWPF value and to compare it with other related polymorphism complexity metrics. The newly proposed polymorphism complexity metric CWPF can be applied and studied for the other object oriented languages. In addition, further empirical studies can be done with software industry groups.

REFERENCES

[1] T. G. Mayer, T. Hall, "Measuring OO systems: a critical analysis of the MOOD metrics," Tools 29, (Procs. Technology of OO Languages & Systems, Europe' 99), R. Mitchell, A. C. Wills, J. Bosch, B. Meyer (Eds.): Los Alamitos, Ca., USA, IEEE Computer Society, pp. 108–117, 1999.

[2] S. Benlarbi, and W. L. Melo, "Polymorphism measures for early risk prediction," IEEE Software Engineering, 1999. Proceedings of the 1999 International Conference, pp. 334-344, 1999.

[3] C. Pons, L. Olsina, and M. Prieto, "A formal mechanism for assessing polymorphism in object-oriented systems," In Quality Software, 2000. Proceedings. First Asia-Pacific Conference on, pp. 53-62. IEEE, 2000.

[4] F. B. Abreu, and W. L. Melo, "Evaluating the impact of object-oriented design on software quality," Proceedings of the 3rd International Software Metrics Symposium (METRICS'96), IEEE, Berlin, Germany, March, 1996.

[5] S. R. Chidamber, C. F. Kemerer, "Towards a metrics suite for object-oriented design," Object-Oriented Programming Systems, Languages and Applications (OOPSLA), vol. 26, pp. 197–211, 1991.

[6] L. Wei, and H. Sallie, "Object-oriented metrics that predict maintainability," Journal of systems and software, vol. 23, no. 2, pp. 111-122, 1993.

[7] F. B. Abreu, and R. Carapuça., "Object-oriented software engineering: Measuring and controlling the development process," Proceedings of the 4th international conference on software quality. vol. 186, pp. 1-8, 1994.

[8] M. Lorenz, and J. Kidd, "Object oriented software metrics," Prentice Hall Object-Oriented Series, Englewood Cliffs, N.J., USA, 1994.

[9] L. H. Rosenberg, and L. E. Hyatt, "Software quality metrics for object-oriented environments," Crosstalk journal, vol. 10, no. 4, pp. 1-16, 1997.

[10] J. Bansiya, and C. G. Davis, "A hierarchical model for object-oriented design quality assessment," IEEE Transactions on Software Engineering," vol. 28, no. 1, pp. 4-17, 2002.

[11] D. Wu, L. Chen, Y. Zhou, and B. Xu. "A metrics-based comparative study on object-oriented programming languages," 2015.

[12] Dufour, Bruno, Karel Driesen, Laurie Hendren, and Clark Verbrugge. "Dynamic metrics for Java," In ACM SIGPLAN Notices, vol. 38, no. 11, pp. 149-168. ACM, 2003.

[13] P. S. Sandhu, and G. Singh, "Dynamic metrics for polymorphism in object oriented systems," World Academy of Science, Engineering and Technology, vol. 2, pp. 03-27, 2008.

[14] Y. Wang, and J. Shao, "Measurement of the cognitive functional complexity of software," Proc. Second IEEE Int. Conf. Cognitive Informatics (ICCI'03), pp. 1-6, 2003.

[15] A. Aloysius, and L. Arockiam, "Cognitive weighted response for a class: A new metric for measuring cognitive complexity of object oriented systems," International Journal of Advanced Research in Computer Science, vol. 3, no. 4, 2012.

[16] A. Aloysisus, and L. Arockiam, "Coupling complexity metric: A cognitive approach," International Journal of Information Technology and Computer Science, vol. 4, no. 9, pp. 29-35, 2012,

[17] F. B. Abreu et al, "The Design of Eiffel Programs: Quantitative Evaluation Using the MOOD Metrics," Proceedings of TOOLS'96, California, Jul. 1996.

[18] N. E. Fenton, and J. Bieman, "Software metrics: A rigorous and practical approach," 3rd edition. CRC Press, ISBN: 9781439838228, pp. 54, November 2014.

[19] F. Thamburaj, "Validation of cognitive weighted method hiding factor complexity metric," in International Conference on Advanced Computing (ICAC 2015), International Journal of Applied Engineering Research (IJAER), accepted for publication.

[20] F. B. Abreu, M. Goulao, and R. Estevers, "Toward the design quality evaluation of object-oriented software systems," Proceedings of the 5th International Conference on Software Quality, Austin, Texas, USA, pp. 44-57. 1995.

T. Francis Thamburaj is working as Assistant Professor in Department of Computer Science, St. Joseph's College, Trichy, Tamil Nadu, India. He has obtained the Master of Computer Applications degree in 1987 and Master of Philosophy degree in 2001 from Bharathidasan University, Trichy. He has 25 years of experience in teaching Computer Science. He is the founder of Computer Science Department in Loyola College, Chennai, in 1993, and Information Technology Department in St. Joseph's College, Trichy, in 2006. His research areas are Artificial Neural Networks and Software Metrics. He has published many research articles in the National / International conferences, and journals. Notably, he has presented, in 2011, a research paper in the World Congress in Computer Science, Computer Engineering, and Applied Computing (WORLDCOMP'11), Las Vegas, USA. A list of his research articles can be found in Google Scholar website. He is currently pursuing Doctor of Philosophy program and his current area of research is Cognitive Aspects of Object Oriented Software Metrics.

A. Aloysius is working as Assistant Professor in Department of Computer Science, St. Joseph's College, Trichy, Tamil Nadu, India. He has got the Master of Computer Science degree in 1996, Master of Philosophy degree in 2004, and Doctor of Philosophy in Computer Science degree in 2013 from Bharathidasan University, Trichy. He has 15 years of experience in teaching and research. He has published many research articles in the National/ International conferences and journals. He has also presented 2 research articles in the International Conferences on Computational Intelligence and Cognitive Informatics in Indonesia. He has acted as a chair person for many national and international conferences. His current area of research is Cognitive Aspects in Software Design.